Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Data & Knowledge Engineering 48 (2004) 75–101
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Adaptive cell-based index for moving objects

Wonik Choi [a,*], Bongki Moon [b], Sukho Lee [a]

[a] *Database Research Laboratory, ENG4190, Seoul National University, Seoul 151-744, South Korea*
[b] *Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA*

Received 4 June 2002; received in revised form 29 January 2003; accepted 7 May 2003

Abstract

R-tree based access methods for moving objects are hardly applicable in practice, due mainly to excessive space requirements and high management costs. To overcome the limitations of such R-tree based access methods, we propose a new index structure called $\mathcal{AIM}$ (*Adaptive cell-based Index for Moving objects*). The $\mathcal{AIM}$ is a cell-based multiversion access structure adopting an overlapping technique. The $\mathcal{AIM}$ refines cells adaptively to handle regional data skew, which may change its locations over time. Through the extensive performance studies, we observed that The $\mathcal{AIM}$ consumed at most 30% of the space required by R-tree based methods, and achieved higher query performance compared with R-tree based methods.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Moving objects; Spatio-temporal databases; Overlapping technique; Multiversion access structure; Cell-based access structure

1. Introduction

Spatio-temporal databases represent, store and manipulate data objects that may change their spatial locations and/or their shapes over time. Due to the geometric and time-varying characteristics, special techniques are often required for efficient storage of and access to spatio-temporal objects.

Recently, there have been many research efforts to develop strategies for indexing and querying moving objects. Among the most fundamental types of queries are *time-slice* or *time-interval* queries, which retrieve all objects that intersect a certain region in space at a specific time point or

[*] Corresponding author.
E-mail addresses: styxii@db.snu.ac.kr (W. Choi), bkmoon@cs.arizona.edu (B. Moon), shlee@cse.snu.ac.kr (S. Lee).

0169-023X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0169-023X(03)00120-4

interval. Another type of common queries is a *trajectory retrieval*, which aims at efficient retrieval of line segments representing trajectories of moving objects. Most of the techniques proposed to handle these types of queries are variants of the R-tree index to process time-slice and time-interval queries [15,26,29] and trajectory retrieval queries [17].

However, we have observed a few critical limitations in the hierarchical access methods. First, an R-tree based index may require high management cost to track mobile objects. The contemporary technologies for global positioning and wireless telecommunications allow us to sample the position of a moving object at discrete instances of time during each sampling period. The sampled positions are transmitted via wireless network to a central server and stored in a database. This amounts to frequent updates to the database for a potentially large number of mobile objects that continuously change their coordinates. Frequent updates made to an R-tree index may lead to deteriorated performance due to a significant amount of overhead to split and merge nodes and increased overlap between minimum bounding rectangles (MBR). For mission-critical applications, this can make it difficult for an R-tree index to keep up with certain operational requirements.

Second, the performance of an R-tree based index tends to deteriorate as the degree of data skewedness increases. Consider a highway traffic monitoring system as an example. A number of vehicles may move along the same paths. This may lead to a few small but extremely dense data regions, where there are many overlapping MBRs, which in turn limits the indexability to prune space and objects during a tree traversal. In this case, an index based on space partitioning seems to be more suitable than that based on data partitioning.

Furthermore, the storage requirement of an R-tree based index is typically high enough to make it less practical to store historical databases, which tend to grow large rapidly. For example, the size of an R-tree that stores trajectories of 1000 objects (about 1.5 million line segments) was about 95MB [17]. It appears quite obvious that the size of an R-tree based index may grow too fast to be used for large-scale applications in practice.

To overcome the limitations of hierarchical access methods, we propose a new index structure called $\mathcal{AIM}$ (*Adaptive cell-based Index for Moving Objects*). The $\mathcal{AIM}$ index consists of two main structures: $\mathcal{MO}$-**Cube** and $\mathcal{MO}$-**Trace**. The $\mathcal{MO}$-**Cube** (Moving Objects—Cube) is a cell-based adaptive index structure to handle spatio-temporal range searches. The $\mathcal{MO}$-**Trace** (Moving Objects—Trace) is a trajectory warehouse that stores time-varying positions of moving objects (i.e., trajectories), and is used for retrieving trajectories after range searches of objects. This two-body structure uses both overlapping and multiversion techniques to produce a time and space efficient index structure, and is particularly useful for processing various spatio-temporal queries.

The $\mathcal{MO}$-**Cube** partitions an embedding space–time into a set of fixed-size grid cells to form a cube that keeps growing along the time dimension. Each grid cell stores a bucket of moving objects in its corresponding subspace. The $\mathcal{MO}$-**Cube** does not change its basic grid cell structure at each sampled time. Instead, it can refine cells in certain regions to handle data skew by adaptively partitioning cells into smaller sub-cells. Such refined cells can appear and disappear anywhere in the grid, as the mobile objects move around in the space and skewed data regions change over time. The rationale of the $\mathcal{MO}$-**Cube** design is to combine the benefits of quadtree (for space decomposition) and hashing method (for bucket storage), without sharing their shortcomings.

To minimize the storage requirement, the $\mathcal{MO}$-**Cube** uses an *overlapping* technique. When a new set of sampled data is added to the $\mathcal{MO}$-**Cube**, new cells can share data buckets with old cells if the positions of the mobile objects in the old cells remain unchanged. Thus, new data buckets are created only when it needs to accommodate changes made to the data set sampled previously.

The $\mathcal{MO}$-**Trace** stores the entire trajectories of moving objects in a space-efficient way. The space efficiency is important because historical databases like such trajectories tend to grow large rapidly.

For the aforementioned reasons, we believe that the $\mathcal{AIM}$ is much better suited for indexing moving objects than R-tree based approaches. Main contributions of this work are:

- To the best of our knowledge, this work is the first attempt to develop a cell-based adaptive index structure for storing and indexing moving objects for time-slice, time-interval and trajectory queries.
- The proposed cell-based adaptive index, augmented with an *overlapping* technique, reduces the storage requirement up to an order of magnitude compared with R-tree based hierarchical access methods.
- This cell-based adaptive index processes spatio-temporal queries for moving objects up to two orders of magnitude faster than R-tree based hierarchical access methods.

The rest of the paper is organized as follows. In Section 2, we survey related work, discuss their advantages and analyze their problems. Section 3 presents the design principles of the $\mathcal{AIM}$ and Section 4 describes the structure of the $\mathcal{MO}$-**Cube** and $\mathcal{MO}$-**Trace** in detail. In Section 5, we provide a detailed description of algorithms for loading and querying. Section 6 contains an extensive experimental evaluation. Finally, Section 7 summarizes the contributions and provides directions for further research.

2. Related work

To store and retrieve the past, current and future locations of moving objects, several research efforts have been reported in the literature. A detailed survey on spatial-temporal databases can be found in [1]. For data models and query language for moving objects see [8,20]. In this section, we briefly survey the previous work, which is mostly based on hierarchical index structures such as R-trees and quadtrees. Depending on the type of data being stored, the following two categories exist: (a) indexing the past and current positions of moving objects and (b) indexing the current positions of moving objects and predicting the future positions based on their current position and motion pattern. Our work belongs to the former category.

2.1. Indexing past and now

The RT-tree [29] couples time intervals with spatial ranges in each node of the tree structure. That is, in addition to its spatial extent, each node contains the time interval during which the corresponding object is alive. The RT-tree adopts ideas from the R-tree and the TSB-tree [13]. Searching or splitting in RT-tree nodes is only guided by the spatial information; the time information plays a secondary role. This makes the RT-tree inefficient, when queries are processed based on the temporal domain, as they would require a complete scan of the index.

The 3D R-tree [26] treats time as another dimension. This is probably the most straightforward way to index spatio-temporal data. The 3D R-tree is very space-efficient since it stores only the different object versions without redundant copies. Since the temporal attributes and the spatial attributes are tightly integrated in 3D R-tree nodes, a time-interval query can be treated as an ordinary window query and thus can be processed very efficiently. However, the performance of time-slice queries can be poor, since all objects are indexed in a single tree and thus the query processing time will depend on the total number of entries in history.

The MR-tree, the HR-tree [15] and the Overlapping Linear quadtree [27] are all based on the concept of *overlapping* trees. Burton et al. [5,6] proposed the concept of *overlapping* trees to index text files evolving over time. Later, the idea was applied to handle the evolving B^+-tree [14]. Recently, this technique was extended to R-trees and quadtrees, to index moving spatial objects [15,27]. The basic idea is that, given two trees where the second one is an evolution of the first one, the second tree can be represented by registering only the modified branches of the first one. That is, only the modified branches are actually created and the branches that do not change are simply re-used, while each tree keeps a root of its own. Fig. 1(a) illustrates an example of the HR-tree.

Since the MR-tree and the HR-tree are very similar in nature, we briefly describes the HR-tree only. The HR-tree can handle a time-slice query very efficiently since time-slice queries are processed inside the corresponding R-tree only. For time-interval queries, however, it can be very slow, because time-interval queries require searching as many trees as the number of distinct timestamps in the interval. Although the HR-tree adopts an *overlapping* techniques, the benefit tends to be limited, because even a single object moving between two consecutive timestamps may cause the replication of multiple nodes. Thus, the size of the HR-tree can be still prohibitive for practical applications.

Recently, Tao and Papadias proposed the MV3R-tree [22], a structure that combines the concepts of multiversion B-trees and 3D R-trees. Fig. 1(b) illustrates the structure of the MV3R-tree. The key idea is to utilize a multiversion R-tree for time-slice queries and a 3D R-tree for time-interval queries. Although the MV3R-tree can deal efficiently with both time-slice and time-interval queries by combining the benefits of both structures, the MV3R-tree requires more space than the 3D R-tree and still needs high management cost due to the combination of the MVR-tree and the 3D R-tree.

We will now examine the access methods that store trajectories as line segments aiming at processing trajectory queries and spatial range queries. Pfoser et al. proposed the STR-tree by adopting three-dimensional R-trees to store line segments of point trajectories bounded by cubes [17]. The STR-tree is an extension of the R-tree to support query processing for the trajectories of

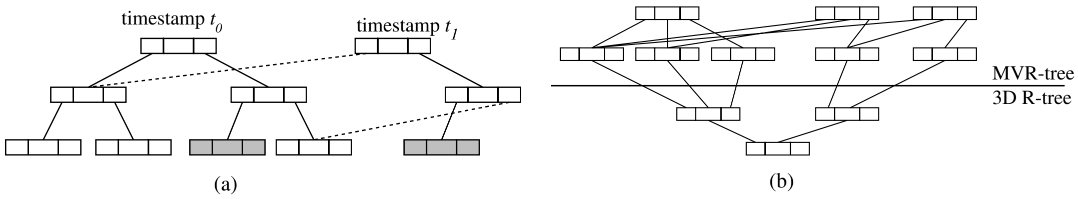


Fig. 1. Structure of (a) HR-tree and (b) MV3R-tree.

moving objects. The insertion procedure in the STR-tree considers spatial proximity and partial trajectory preservation. That is, the STR-tree tries to keep line segments belonging to the same trajectory together. Also, the authors proposed the TB-tree, which aims for an access method that strictly preserves trajectories such that a leaf node only contains line segments belonging to the same trajectory. Although the TB-tree supports trajectory queries much more efficiently than the R-tree does, its performance on time-slice and time-interval query is worse than the R-tree for a large number of moving objects since the TB-tree does not consider spatial discrimination. As for the STR-tree, although designed to combine the benefits of the TB-tree and the R-tree, it usually performs worse than the TB-tree and is rather a weak compromise.

2.2. Indexing now and future

There have been quite a few related research efforts on the indexing of the current and anticipated future positions of moving objects. Most of the work aimed at reducing update cost, and used a simple linear function [18,23] or a data transformation [2,11] to describe the movements of moving objects. We will briefly survey the work which is closely related to our approach.

Song and Roussopoulos proposed an access method based on a hashing technique that stores the bucket information of each object instead of its exact location [21]. This index structure remains unchanged until an object moves into a new bucket. Therefore, the update cost is greatly reduced. However, this method does not make any provision for handling queries other than the current locations. Kwon et al. also aim at indexing the current positions only [12]. They proposed the LUR-tree (Lazy Update R-tree) to reduce update cost by modifying the index structure only when an object moves out of the corresponding MBR.

More recently, Chon et al. have developed a space–time partitioning model where the trajectory of a moving object is modeled as a polyline [7]. In reality, the trajectory of a moving object is the result of the interactions among all moving objects. They took these factors into account, and thus aimed at the better management of dynamic information about moving objects and accurate prediction of their trajectories. This work was based on a grid as we do, but focused on querying the future positions of moving objects.

3. Design of the $\mathcal{AIM}$

In this section, we describe what assumptions we make and present a sample application scenario. Types of queries the proposed $\mathcal{AIM}$ index aims at are also described.

3.1. Design principles

Theodoridis et al. proposed a specification and classification scheme for spatio-temporal access methods (STAMs), and suggested a list of specifications that can be followed by an STAM [24]. They addressed issues on data types and characteristics, index construction, and query processing operations. A few existing proposals such as RT-tree, HR-tree and 3D R-tree were evaluated according to the specifications. We adopt the same scheme to precisely describe the specifications of the $\mathcal{AIM}$ index.

Table 1 summarizes the specifications of the $\mathcal{AIM}$ and compares with other existing STAMs. The $\mathcal{AIM}$ index considers `point` objects moving in a discrete manner within a data space. In this paper, we assume that a point moves in a two-dimensional space; our approach can be easily expanded into a three-dimensional space. The support of `region` data type needs further investigation and we leave it as future work. We support `transaction-time`. The timestamp of each point increases monotonically following a transaction-time pattern. Since both the cardinality of the data set and the object locations may change with time, the $\mathcal{AIM}$ is classified as `full-dynamic`. In addition, only current instances are dynamically inserted into the $\mathcal{AIM}$ index (`chronological`).

All STAMs in Table 1 except the $\mathcal{AIM}$ are based on R-trees and variants, which approximate spatial objects by their MBRs. Since objects move around in the work space, their MBRs tend to include considerable dead spaces and overlap each other. This fact in turn leads to inefficient indexing. In contrast, the $\mathcal{AIM}$ does not use MBRs at all, but instead it stores an exact location of each moving object. Note that in the `Spec6` entry of Table 1, the $\mathcal{AIM}$ is the only index that efficiently handles a purge operation for *obsolete* objects. In fact, R-tree based index structures have been known for their lack of efficient mechanisms for deleting obsolete objects [10].

3.2. An application scenario

With the rapid and continued advances of positioning systems such as GPS and wireless communication technologies, tracking and recording the changing positions of objects are becoming increasingly feasible. The need for indexing moving objects and processing various type of spatial-temporal queries arises in a wide range of moving-objects-database (MOD) applications, including traffic supervision systems and transportation systems in the civilian industry, digital battlefields in the military [28]. As an example consider the traffic supervision systems, which monitor the location and motion patterns of vehicles in order to provide services such as congestion detection, optimal route guidance and so on. Each vehicle is equipped with a GPS device, and automatically and periodically transmits its position to a server at regular intervals using either radio communication link or cellular phones. The server stores these current positions of all vehicles which constitute the trajectory of moving object, and processes spatial-temporal queries. Our objective is to minimize the management cost and the storage requirement for such an application. In addition, we aim at efficient processing of the following types of queries: *time-slice query*, *time-interval query*, and *trajectory query*.

- *Time-slice query* retrieves all moving objects within a certain region at a specific time *slice*.
- *Time-interval query* retrieves all moving objects within a certain region at a specific time *interval*.
- *Trajectory query* retrieves (partial) trajectories of moving objects that satisfy a certain condition specified in a spatio-temporal domain. For example, "For all flying objects that passed through the Area-59 between 8:30 am and 9:00 am on November 2, 2001, find their trajectories during the period from 9:00 am till 10:00 am on the same date."

Typically, trajectory queries can be processed in two steps. The first step is to retrieve object IDs based on a spatio-temporal condition, such as "Area-59 between 8:30 am and 9:00 am on

Table 1
Evaluation of the $\mathcal{AIM}$ and existing STAMs

	HR-tree	3D R-tree	TB/STR-tree	MV3R-tree	$\mathcal{AIM}$
Spec1: *Data types supported*	`region`	`region`	`region`	`region`	`point`
Spec2: *Type of time supported*	`transaction`	`valid`	`valid`	`transaction`	`transaction`
Spec3: *Data set mobility*	`full-dynamic`	`growing`	`full-dynamic`	`full-dynamic`	`full-dynamic`
Spec4: *Time-stamp update*	`chronological`	`static`	`chronological`	`chronological`	`chronological`
Spec5: *Specific object approximation*	`YES`	`NO`	`YES`	`YES`	`NO`
Spec6: *Handling ‘obsolete’ entries*	`NO`	`NO`	`NO`	`NO`	`YES`
Spec7: *Specific query processing operations*	`YES`	`YES`	`YES`	`YES`	`YES`

November 2, 2001" in the example. In the second stage, the corresponding trajectories of the objects are retrieved by limiting the scope of the trajectories, with an additional temporal range such as "during the period from 9:00 am till 10:00 am on the same date."

4. The structure of the $\mathcal{AIM}$

Formally, in our proposed $\mathcal{AIM}$ index structures, moving objects are modeled within grid cells. A moving object M moves in a three-dimensional space $X \times Y \times T$, where $X = [x_0, x_{n-1}]$, $Y = [y_0, y_{n-1}]$ and $T = [t_0, t_{p-1}]$. The three-dimensional space is partitioned into grid cells such that

$$X = [x_0, x_1) \cup [x_1, x_2) \cup \cdots \cup [x_{n-2}, x_{n-1}],$$

$$Y = [y_0, y_1) \cup [y_1, y_2) \cup \cdots \cup [y_{n-2}, y_{n-1}],$$

$$T = [t_0, t_1) \cup [t_1, t_2) \cup \cdots \cup [t_{p-2}, t_{p-1}],$$

where t_p means *now* timestamp. The lifespan of each cell is $[t_i, t_{i+1})$.

The position of M at a certain point in a time dimension T is a point in a three-dimensional space $X \times Y \times T$. Thus, the trajectory of M is a polyline in a three-dimensional space $X \times Y \times T$. The part of the trajectory is defined as follows:

$$m(u) = \bigcup_{i=0}^{q} l_i, \tag{1}$$

where $l_i = \{\langle x_i, y_i, t_i \rangle, \langle x_{i+1}, y_{i+1}, t_{i+1} \rangle\}$ and u is an identifier of group of line segments. An element $\langle x_i, y_i, t_i \rangle$ is the ith version of position of $M(id)$. Thus, the trajectory of $M(id)$ is a set of groups of line segments $m(u)$, and is modeled as follows:

$$M(id) = \bigcup_{u=0}^{v} m(u). \tag{2}$$

When we refer to a specific grid cell $[x_i, x_{i+1}) \times [y_j, y_{j+1})$ at a certain time $[t_k, t_{k+1})$, we use the notation $g[c]_k$, which c is a cell number from an address function $G(x_i, y_j)$.

A grid cell $g[c]_k$ is modeled as a set of $\langle id, u \rangle$:

$$g[c]_k = \{\langle id, u \rangle | u \in U, id \in ID, \text{ such that } \exists l \text{ in } m(u) \text{ intersects } g[c] \text{ during } [t_k, t_{k+1})\}, \tag{3}$$

where U is a set of identifier of group of line segments and ID is a set of unique identifier of moving objects.

Based on the grid model, we build two data structures $\mathcal{MO}$-**Cube** and $\mathcal{MO}$-**Trace**, which constitute the $\mathcal{AIM}$ index. This *two-body* design of the $\mathcal{AIM}$ index is particularly useful for processing trajectory queries, because trajectory queries are typically processed in two stages as shown in the previous section. That is, the $\mathcal{MO}$-**Cube** is used in the first stage, and the $\mathcal{MO}$-**Trace** in the second stage. Of course, the $\mathcal{MO}$-**Cube** alone can process spatial-temporal range searches efficiently.

4.1. $\mathcal{MO}$-Cube

4.1.1. $\mathcal{MO}$-Cube as a pile of slices

The $\mathcal{MO}$-**Cube** is a collection of $\mathcal{MO}$-**Slice**s, each of which serves as an index for a set of moving objects at a specific time interval. Fig. 2 illustrates the structure of the $\mathcal{MO}$-**Cube**. Each bucket of $\mathcal{MO}$-**Slice** consists of three parts: *Bucket Set*, *Time Array* and *Refinement Array*. The *Bucket Set* stores the buckets corresponding timestamps, and is implemented as an *append-only* file with a consecutive set of disk pages. When new versions of the position of moving objects are sampled, the entries for each moving objects are added to its corresponding cell (or bucket). All those buckets of the $\mathcal{MO}$-**Slice** then are collectively appended to the corresponding *Bucket Set*. Each entry for a moving object in a bucket has the form $\langle id, ptr \rangle$ as shown in Eq. (3), where an *id* is the identifier of the moving object and a *ptr* points to a page of the $\mathcal{MO}$-**Trace**, which contains the line segments intersected with the bucket.

The *Time Array* represents a time domain of a bucket, and is used for finding a bucket corresponding to its grid cell at a specific timestamp. The *Time Array* contains offsets of buckets within a *Bucket Set*, and is represented by a one-dimensional array. The *Refinement Array* is an array that stores the metadata of *cell refinement*. The details of the cell refinement will be discussed in Section 4.1.2.

Initially, a $\mathcal{MO}$-**Slice** is partitioned to $C(=n \times n)$ cells. The choice of C value may have non-trivial impact to the performance of the $\mathcal{AIM}$ index in storage utilization and query processing. Bentley et al. show that a nearly optimum size for the cells are the same as those of the query window size [4]. In most applications, however, the queries will vary in size as well as in location, so there is little information available for making a good choice of cell size. Thus we propose the following alternatives to choose C: (1) one based on the population of moving objects, and (2) the other based on the maximum distance that an object can move during a given time period.

The C value can be determined rather intuitively by identifying an ideal number of objects per cell. Suppose there are a total of N moving objects to index, and a disk page can store up to E entries. Then, C can be chosen to be

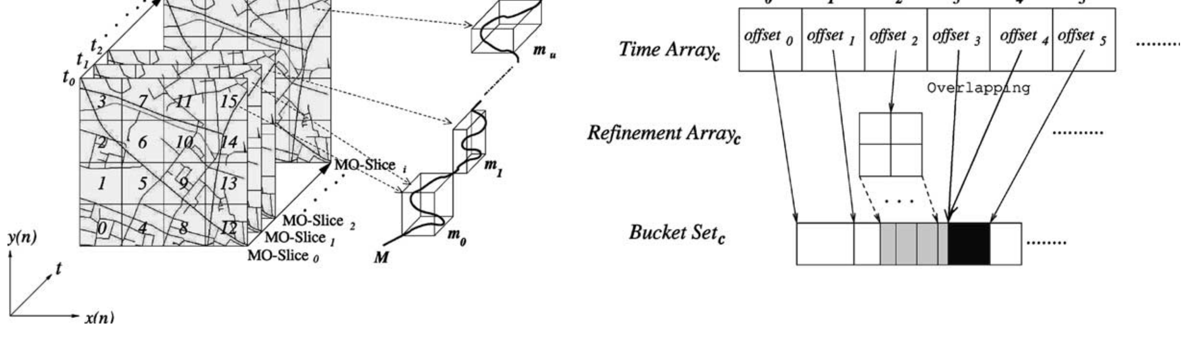


Fig. 2. Structure of the $\mathcal{MO}$-**Cube**.

$$C = \left(\left\lceil \sqrt{\frac{N}{E}} \right\rceil \right)^2. \tag{4}$$

Alternatively, if the maximum velocity of all moving objects is known, they can be used to determine the C value. The idea is that if a moving object stays in the same cell for a longer period of time, then it is more probable that the bucket of the cell may be reused by an *overlapping* technique. That is, C can be set to the maximum distance an object can move between two consecutive timestamps. The overlapping technique will be discussed in Section 4.1.3.

4.1.2. Adaptive cell refinement

Mobile objects are dynamic by nature. The distribution of mobile objects is not predictable and may be skewed. Data skew can cause a data bucket in a dense region to overgrow in size, which in turn can lower the efficiency in data accesses. To address the data skew problem, we refine grid cells adaptively depending on the data population of cells.

If the number of mobile objects in a cell exceeds E (the maximum number of entries per cell) and its bucket overflows, the overcrowded cell is partitioned recursively and the mobile objects in the cell are *split* across the sub-cells until there is no sub-cell whose population is larger than E. Note that there is no explicit *merge* operation defined for the $\mathcal{MO}$-**Cube**, because the $\mathcal{MO}$-**Cube** is designed to be an append-only index structure. Fig. 3 illustrates the adaptive cell refinement. At time t_0, none of the cells are overcrowded and refined. On the other hand, cell 6 and cell 9 become overcrowded at time t_1 and t_2, respectively. Thus, cell 6 is partitioned to 4 sub-cells, and cell 9 is partitioned to 16 sub-cells. When a cell is refined, its entry in the *Time Array* stores a pointer to a corresponding entry in the *Refinement Array* and the level of cell refinement, instead of a pointer to a data bucket.

By adaptively refining grid cells, we can accelerate query processing for time-slice and time-interval range queries. When a query region overlaps a sub-cell, only a part of a data bucket needs to be accessed to answer the query without retrieving all moving objects in the data bucket.

Fig. 3(b) shows the storage structure of the $\mathcal{MO}$-**Cube** example given in Fig. 3(a). The *Time Array*$_6$ for cell 6 at time t_0 points directly to the corresponding data bucket, because cell 6 is not refined. However, the *Time Array*$_6$ at time t_1 points to the *Refinement Array*$_6$ instead, because cell 6 is refined to 4 sub-cells. Each entry in the *Refinement Array*$_6$ points to a data bucket corresponding to a sub-cell. In a similar way, the *Time Array*$_9$ for cell 9 at time t_2 points to the 16

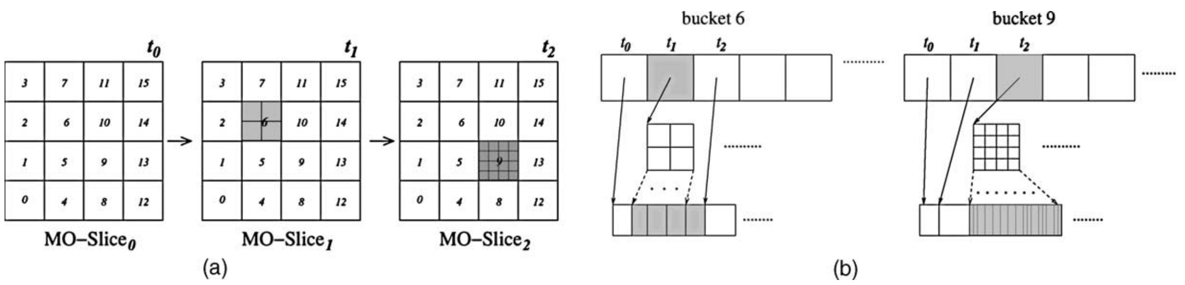


Fig. 3. Adaptive Cell Refinement.

entries in the *Refinement Array*$_9$. By using the *Refinement Array*$_9$, the adaptive cell refinement technique can help reduce query processing cost significantly.

4.1.3. Overlapping the $\mathcal{MO}$-Slices

The $\mathcal{MO}$-**Cube** adopts an *overlapping* technique [5,6]. For each cell, several $\mathcal{MO}$-**Slice** instances are constructed at different timestamps. However, in order to save disk space, common data buckets of two consecutive $\mathcal{MO}$-**Slice**s may be combined without data replication, if the contents of two buckets are identical. In the example shown in Fig. 4, only one (gray) of the four buckets has changed. Hence, the $\mathcal{MO}$-**Slice**s at timestamp t_0 and t_1 in Fig. 4(a) and (b) can be stored in a more compact manner as shown in Fig. 4(c). The implementation of overlapping $\mathcal{MO}$-**Slice**s is also illustrated in the right side of Fig. 2. For the cell number c, both the *offset*$_3$ and *offset*$_4$ (for timestamps t_3 and t_4, respectively) point to the same bucket in the *Bucket Set*, because the entries in the cell remain unchanged.

In addition to the *overlapping* technique, a *multiversion* technique has been applied on various index structures such as B-tree, R-tree and linear quadtrees to support versioning databases. These index structure adopt *overlapping* or *multiversion* technique may share common branches in order to avoid excessive space requirements. To quantify the space savings from the *overlapping* and *multiversion* technique, we define the *reuse rate* in the practical perspective of the space savings as follows:

$$R = \left(1 - \frac{|T_i|}{|T_i'|}\right) * 100 \quad (|T_i| \leqslant |T_i'|), \tag{5}$$

where $|T_i|$ is the index size with the *overlapping* or *multiversion* technique at timestamp i and T_i' without the *overlapping* or *multiversion* technique. This formula will be used to measure space savings in Section 6.5.

4.1.4. Discussion

The basic idea of the $\mathcal{MO}$-**Cube** is similar to that of the quadtree [9,19], in the sense that it is based on the principle of recursive space decomposition. An important difference, however, is the fact that the quadtree, basically, is a main memory structure which is less suited for large spatio-temporal databases. Moreover, since the quadtree is based on unbalanced n-ary trees, subtrees corresponding to densely populated regions may be deeper than others, which in turn may incur a significant degradation of query performance.

From another point of view, the structure of the $\mathcal{MO}$-**Cube** is also similar to that of the gridfile [9,16]. The main difference between the $\mathcal{MO}$-**Cube** and the gridfile is the fact that the gridfile has a

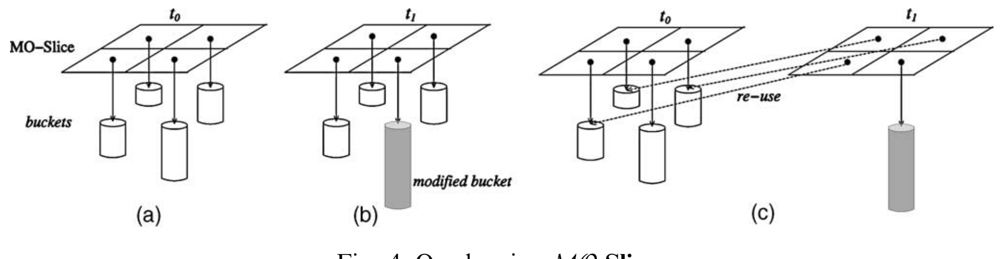


Fig. 4. Overlapping $\mathcal{MO}$-**Slices**.

grid directory and suffers from a super linear growth of the grid directory even for uniformly distributed data sets. As mentioned briefly in Section 1, the rationale of the $\mathcal{MO}$-**Cube** design is to combine benefits of the quadtree and the gridfile without inheriting any of their shortcomings.

One of the most important properties that multidimensional access methods should possess is the independence of the distribution of data sets. Multidimensional access methods should provide a facility to maintain their efficiency even when the data sets are highly skewed. From this point of view, it is true that hierarchical access methods such as R-trees can handle highly skewed data sets more efficiently than hashing based methods can. At the same time, it is worth mentioning that its storage utilization of nodes is higher than that of data buckets for hashing methods. The $\mathcal{MO}$-**Cube** may also have these problems, too. But their effects may be mitigated by the methods presented in the previous subsections. First, in order to alleviate the problem stemming from skewed data sets, the $\mathcal{MO}$-**Cube** employs the adaptive cell refinement scheme as described in Section 4.1.2. As the results from our experiments will show, this scheme can help handle highly skewed data sets in a satisfactory way. In addition, the $\mathcal{MO}$-**Cube** is augmented with the overlapping technique as described in Section 4.1.3 in order to improve storage utilization.

4.2. $\mathcal{MO}$-Trace

Although R-tree based methods are known for their effectiveness in indexing for a wide spectrum of spatial database applications, they are not particularly well suited for trajectory queries, because the trajectory queries require keeping track of chronological positions of given mobile objects at each point in time. It would be quite costly if an R-tree index should be traversed to find each position of an object. To avoid such disadvantages of R-tree based methods, we take an entirely different approach called $\mathcal{MO}$-**Trace** as shown in Fig. 5.

The $\mathcal{MO}$-**Trace** is a simple but elegant storage structure that is designed to be space-efficient and capable of retrieving a sequence of chronological positions of a mobile object in a straight-

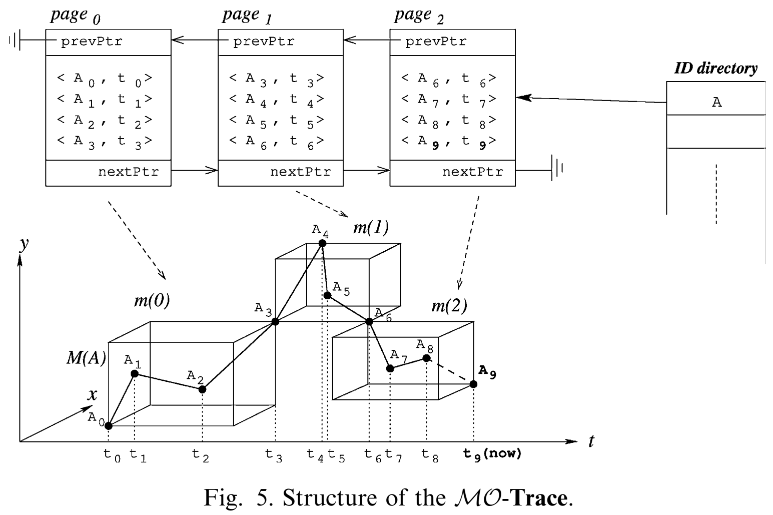


Fig. 5. Structure of the $\mathcal{MO}$-**Trace**.

forward manner. The $\mathcal{MO}$-**Trace** implements the model of the trajectory which is shown in Eqs. (1) and (2). A new version of the position of moving objects obtained from data sources such as GPS is appended to the page which is pointed to by the *ID directory*. For given objects and their IDs, the *ID directory* is used to find the current pages for them. Then the sampled positions of objects are written to the current pages. The *ID directory* can be implemented using one-dimensional array or hash tables. Each entry in a page has the form $\langle id_i, t_i\rangle$ where id_i denotes the position of an object at timestamp t_i. The two consecutive entries constitute a line segment, i.e., $\langle id_i, t_i\rangle - \langle id_{i+1}, t_{i+1}\rangle$. These line segments constitute a group of line segments, which correspond to a disk page as shown in Fig. 5. We choose a doubly linked list that connects pages including segments of the same trajectory. Therefore, in addition to the entries, each page has elements, such as `prevPtr` and `nextPtr`, which are used for the linked list. One can retrieve *forward or backward* trajectories by following the pointers of the linked list from a page pointed to by an entry of the $\mathcal{MO}$-**Cube**, i.e., $\langle id, ptr\rangle$ to the next or previous pages. The $\mathcal{MO}$-**Trace** is a pool of pages that are connected by a doubly linked list for the trajectory of moving objects. Also, the $\mathcal{MO}$-**Trace** may be used in combination with R-tree based methods if entries of R-tree base methods are slightly modified to contain the pointer to a page of the $\mathcal{MO}$-**Trace**.

Although the $\mathcal{MO}$-**Trace** does not explicitly store dynamic information such as the average or maximum speed and heading, it can be computed by extracting a sequence of positions between two specified timestamps.

5. Algorithms for loading and querying

5.1. Inserting moving objects to $\mathcal{AIM}$

In this section, we describe how to insert $M(id)$ into the $\mathcal{AIM}$. The simplest approach to represent the trajectories of moving objects would be to store each sampled position into the corresponding bucket. To answer queries about trajectories at times in-between those of the sampled positions, we treat the sampled positions as the endpoints of line segments of polylines in a 3D space. We will use the following example in Fig. 6 to describe how to insert a line segment into the $\mathcal{AIM}$.

For the sake of simplicity of our example, we assume that a page in the $\mathcal{MO}$-**Trace** can accommodate 3 entries and each $\mathcal{MO}$-**Slice** is partitioned to 4 cells. The initial situation of insertion procedure is given in Fig. 6(a). At timestamp t_0, A_0 and B_0, the current positions of $M(A)$, $M(B)$, are inserted into the current page of the $\mathcal{MO}$-**Trace**. Then, the pointer to the current page of the $\mathcal{MO}$-**Trace** and an *id* are written to the corresponding bucket of the $\mathcal{MO}$-**Cube**. For example in Fig. 6(a), A_0 belongs to cell 1 and is inserted into the page p_0 of the $\mathcal{MO}$-**Trace**. Thus, an entry $\langle A, p_0\rangle$ is inserted into the bucket 1 of the $\mathcal{MO}$-**Cube** at t_0. We call this procedure $\mathcal{P}$*ointInsertion*.

At the next sampling time t_1, new positions are inserted into the $\mathcal{MO}$-**Trace** and the $\mathcal{MO}$-**Cube** in the same way as described above. In addition, since the two points such as $\langle A_0, t_0\rangle$ and $\langle A_1, t_1\rangle$ constitute a line segment, the information of the line segment should also be inserted into the corresponding bucket at timestamp t_0. By tracking the line segment in all the cells they pass through, we can answer time-slice/interval queries even at a time point/interval between two sampled positions by interpolating them.

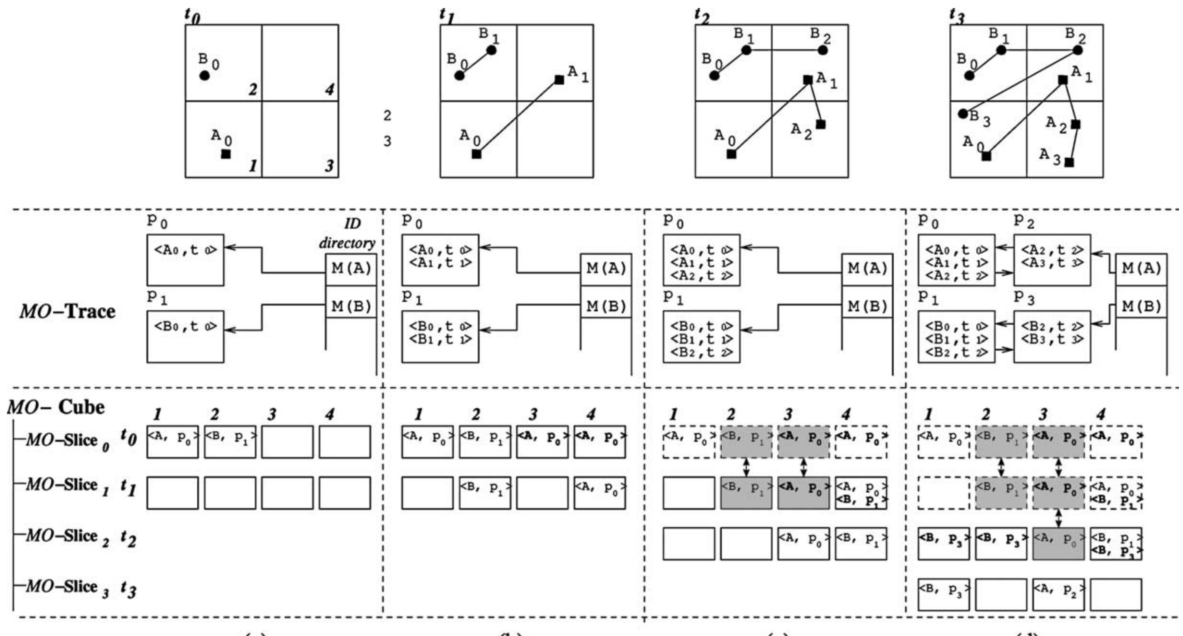


Fig. 6. Insertion.

Each entry in bold-face fonts such as $\langle A, p_0 \rangle$ in a bucket of $\mathcal{MO}$-**Slice** at t_0 in Fig. 6(b) indicates that a line segment one of whose end points is $\langle A, p_0 \rangle$ has been added. Specifically, since a line segment with end points $\langle A_0, t_0 \rangle$ and $\langle A_1, t_1 \rangle$ intersects cells 1, 3 and 4 during a time period $[t_0, t_1)$, each of the buckets 1, 3 and 4 of the $\mathcal{MO}$-**Cube** at t_0 contains the page pointer p_0 of the $\mathcal{MO}$-**Trace**. We call this second procedure *LineInsertion*.

After the *LineInsertion* procedure, the buckets at t_1 are now ready to be appended to the *Bucket Set* of the $\mathcal{MO}$-**Cube**. This means that the *overlapping* technique can now be adopted to these buckets. When the buckets are written to the *Bucket Set*, a bucket identical to one of the previous buckets in the *Bucket Set* is not actually written to disk. Instead, it shares the content with the previous bucket. The gray buckets of the $\mathcal{MO}$-**Cube** in Fig. 6(c) and (d) show the *overlapped* buckets. We call this procedure *Overlapping*.

In summary, the insertion procedure of the $\mathcal{AIM}$ consists of three stages. In the first stage, the *PointInsertion* procedure which simply inserts a point into the corresponding bucket is performed. Next, the *LineInsertion* procedure appends the information of the line segment to the previous buckets. After the *LineInsertion* procedure, the *Overlapping* procedure writes the buckets to the *Bucket Set* of the $\mathcal{MO}$-**Cube**. During this procedure, the buckets which are identical with the previous buckets are not actually written to a disk, but share the previous bucket in the *Bucket Set*. The dotted line boxes of the $\mathcal{MO}$-**Cube** in Fig. 6(c) and (d) mean the buckets that are already moved to the *Bucket Set* in this way.

To handle these three procedures efficiently, each bucket has two temporary pages called *collector page*. The solid line boxes of the $\mathcal{MO}$-**Cube** in Fig. 6 show the collector pages of the corresponding bucket. The first collector page contains entries at *now* and the second collector page contains entries at $now-1$. After the collector page at $now-1$ is written to the *Bucket Set* at $now-2$, the collector page is reused for the collector page at *now*.

Putting it together, we get the insertion procedure of the $\mathcal{AIM}$ shown in Algorithm 1. Note that the $\mathcal{R}$*efineCell* and the $\mathcal{O}$*verlapping* procedure are invoked by the $\mathcal{P}$*ointInsertion* procedure. If the position whose timestamp is larger than the bucket's lifespan is arrived, a new $\mathcal{MO}$-**Slice** is created and the $\mathcal{O}$*verlapping* procedure is invoked (line 8). Otherwise, the position whose timestamp is covered by the lifespan of the bucket is written to the corresponding buckets. At this time if the bucket is full, the $\mathcal{R}$*efineCell* procedure is invoked (line 9).

Algorithm 1. Insertion Algorithm for the $\mathcal{AIM}$

```
     Procedure Insertion ($id, x, y, t$)
     begin
1:     invoke $\mathcal{P}ointInsertion(id, x, y, t)$;
2:     invoke $\mathcal{L}ineInsertion(id, x, y, t, x_{-1}, y_{-1}, t_{-1})$;
     end
     Procedure $\mathcal{P}ointInsertion(id, x, y, t)$
     begin
3:    $PagePtr \leftarrow$ GetPagePointer($id$);    // Get the current page from the $\mathcal{MO}$-Trace.
4:    WritePosition($PagePtr, x, y, t$);         // Write the position to the $\mathcal{MO}$-Trace.
5:    $c \leftarrow$ FindBucket($x, y, t$);    // Find the bucket which a position belongs to.
6:    if $Bucket[c]$ has not two collector pages then
          Allocate two collector pages $Collector[c][0]$ and $Collector[c][1]$ to $Bucket[c]$.
          let $NowPage[c] \leftarrow Collector[c][0]$.
          let $PrevPage[c] \leftarrow Collector[c][1]$.
7:    if $NowPage[c].t_{end} < t$ // if the lifespan of bucket is over, the bucket is now overlapped. then
8:      Invoke $\mathcal{O}verlapping(c)$;
      else
9:    if $NowPage[c]$ is full then invoke $\mathcal{R}efineCell(NowPage[c])$.
10:   WritePagePtr($PagePtr$, $NowPage[c]$);   // Append the $PagePtr$ to $NowPage[c]$.
      end
     end
     Procedure $\mathcal{L}ineInsertion(id, x, y, t, x_{-1}, y_{-1}, t_{-1})$
     begin
11:    $PagePtr \leftarrow$ GetPagePointer($id$);
12:    $\mathcal{B}ucketSet\{c\} \leftarrow$ Intersect($x, y, t, x_{-1}, y_{-1}, t_{-1}$);
13:    for each $c \in \mathcal{B}ucketSet\{c\}$ do
14:      if $NowPage[c]$ is full then invoke $\mathcal{R}efineCell(PrevPage[c])$.    // Refine a cell adaptively.
15:      WritePagePtr($PagePtr$, $PrevPage[c]$);
       end
     end
     Procedure $\mathcal{O}verlapping(c)$
     begin
16:    if the $PrevPage[c]$ equals to the previous bucket in the $\mathcal{MO}$-Cube then
17:      Reuse the previous bucket.
```

```
          else
18:         Append the PrevPage[c] to the Bucket Set.
          end
19:       let NowPage[c] ← Collector[c][1].   // Swap NowPage with PrevPage.
          let PrevPage[c] ← Collector[c][0].
20:       Clear NowPage[c].   // for reuse.
      end
      Procedure RefineCell(Page)
      begin
21:       Page.level++;
22:       Partition the Page to $2^{Page.level}$ sub_Pages.
23:       for each $c \in Page$ do
            Reinsert c into sub_Page.
          end
      end
```

5.2. *Query processing*

5.2.1. *Time-slice and time-interval queries*

One of the fundamental objectives of the $\mathcal{AIM}$ is to efficiently handle time-slice and time-interval queries. The cell-based structure of the $\mathcal{MO}$-**Cube** makes the query processing for time-slice and time-interval queries intuitive and straightforward. Time-slice and time-interval query processing involve selecting a set of entries of corresponding to the timestamp interval in question. The $\mathcal{MO}$-**Cube** uses the address function $G(x,y)$ in Eq. (6) below to find a grid cell which a moving object moves in. The address function G returns a pointer to an entry in the *Time Array* corresponding to a grid cell that contains (x,y). The function G is defined as

$$G(x,y) = \left\lfloor \frac{x}{w} \right\rfloor (\sqrt{C} - 1) + \left\lfloor \frac{y}{w} \right\rfloor, \tag{6}$$

where w is the width of a cell, and C is the total number of cells (see Fig. 2).

Suppose the timestamp interval of interest is from t_s to t_e. A search operation is performed by reading the corresponding $\mathcal{MO}$-**Slice** from t_s to t_e. Then, a set of the buckets intersected with a query window R is obtained. Specifically, in line 4 of Algorithm 2, the *CandidateSet* is generated as a set of $\langle id, PagePtr \rangle$. Note that since an entry $\langle id, PagePtr \rangle$ is most likely to be inserted into multiple buckets, we must eliminate the duplicated entries in order to have only one copy of an entry. The next job is to check if line segments in *PagePtr* are actually intersected with R (in line 6 of Algorithm 2). We call this procedure an *intersection test*. This algorithm can support both time-slice query and time-interval query.

As an example, consider a query which is looking for all moving objects in some region overlapping the query window q_s of Fig. 7 during time interval $[t_1, t_2]$. Search starts by reading the *Time Array* of the $\mathcal{MO}$-**Slice**. Since cell 2 and 4 intersect q_s, the *CandidateSet* is $\{\langle A, p_0 \rangle, \langle B, p_1 \rangle, \langle B, p_3 \rangle\}$ after reading the bucket 2 and 4 during $[t_1, t_2]$, which are shown as gray boxes of the $\mathcal{MO}$-**Cube** in Fig. 7. In order to complete the result, the *intersection test* checks whether the line segment in

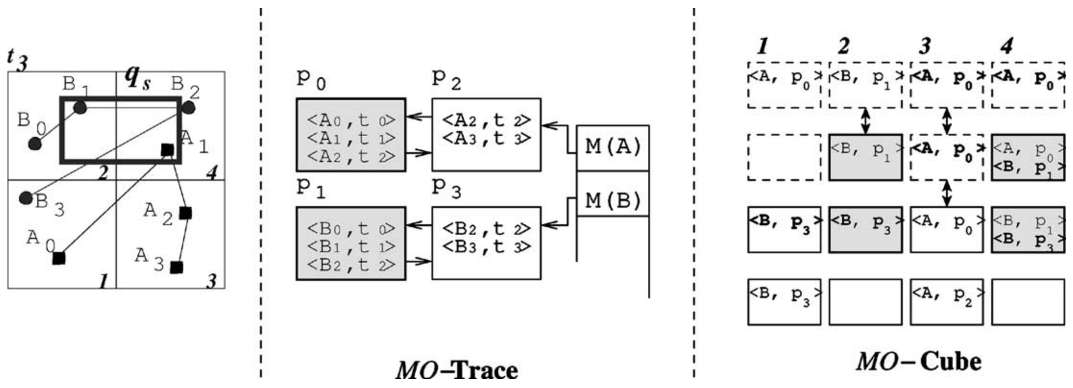


Fig. 7. Example of Query Processing.

PagePtr actually intersects q_s. Since the line segment in p_0 intersects q_s, $\langle A, p_0\rangle$ is included in the *AnswerSet*. In the same way, $\langle B, p_1\rangle$ is also included in the *AnswerSet*. But, $\langle B, p_3\rangle$ is excluded since B is already included in the *AnswerSet*. Therefore, the final result is $\{\langle A, p_0\rangle, \langle B, p_1\rangle\}$.

Algorithm 2. Time-slice/interval Query Algorithm

```
    Procedure Time-IntervalQuery ($t_s, t_e, R$)
    // Given a query window $R$ and a time interval,
    // Find all moving objects within R at any time
    // between $t_s$ and $t_e$ ($t_s < t_e \leqslant t_{now}$)
    begin
1:      set CandidateSet ← an empty set;
2:      set AnswerSet ← an empty set;
3:      for each MO-Slice between [$t_s, t_e$] do
4:          CandidateSet{c} ← Intersect(R);    //Get id and PagePtr intersected with R.
        end
5:      for each $id \in$ CandidateSet[c]   // intersection test do
6:          if $id \notin$ AnswerSet and any line segments in PagePtr intersect R then
7:              AnswerSet ← ⟨id, PagePtr⟩ ∪ AnswerSet;
            end
        end
    end
```

5.2.2. Trajectory retrieval

The retrieval of trajectories is shown in Algorithm 3. After we get the moving object ID and *StartPage* from the *MO*-**Cube**, we can easily retrieve the corresponding trajectories from the *MO*-**Trace** as shown in Algorithm 3. First, we read the doubly linked list of *id* backward in order to get the trajectories from t_s to the latest timestamp in *StartPage* (Algorithm 3—line 2–4). Then, we read pages forward in order to get the trajectories from the latest timestamp in *StartPage* to t_e (Algorithm 3—line 6–8).

As an example, consider Fig. 7 in the previous section. Since the *AnswerSet* is obtained after time-interval query, the trajectories can be retrieved by reading the page which *StartPage* points out, i.e., p_0 and p_1. As a result, the trajectories of A and B will be $\{\langle A_1, t_1\rangle, \langle A_2, t_2\rangle\}$ and $\{\langle B_1, t_1\rangle, \langle B_2, t_2\rangle\}$, respectively.

5.3. Purge operation

Purge operation to handle '*obsolete*' entries is very important in spatio-temporal access methods, because maintaining all versions of each timestamp may be too costly in terms of space requirements. In order to reduce space requirements, index pages contains '*obsolete*' entries can be purged. The *purge* operation of R-tree based methods removes pages consisting of '*obsolete*' entries from the tree structure. This *purge* operation actually leads to unbalanced trees because several nodes are removed without affecting the rest of nodes and pointers. Becker et al. addressed this problem in [3]. However, the *purge* operation of the $\mathcal{AIM}$ is very simple and intuitive. The $\mathcal{AIM}$ removes the $\mathcal{MO}$-**Slice**s and the part of doubly linked lists of the $\mathcal{MO}$-**Trace** corresponding to '*obsolete*' timestamps to a secondary or tertiary storage. The restoration of '*obsolete*' the $\mathcal{MO}$-**Slice**s and the doubly linked lists is simple as well.

Algorithm 3. Retrieval of trajectory

```
   Procedure TrajectoryRetrieval ($id, StartPage, t_s, t_e$)
   // Given a moving object ID and a StartPage which retrieval starts from,
   // Retrieve the trajectory of a moving object ID
   // at any time between $t_s$ and $t_e$ ($t_s < t_e \leqslant t_{now}$)
   begin
1:   set $Page \leftarrow StartPage$;
     while Page is not NULL do
2:      Read line segments from Page.
3:      if Page contains line segment at $t_s$ then break
4:      $Page \leftarrow Page.PrevPtr$;
     end
5:   set $Page \leftarrow StartPage$;
     while Page is not NULL do
6:      Read line segments from Page.
7:      if Page contains line segment at $t_e$ then break;
8:      $Page \leftarrow Page.NextPtr$;
     end
   end
```

6. Performance evaluations

In this section, we evaluate the $\mathcal{AIM}$ empirically and compare with the previous work through extensive experimentation. The data set we used are described, and the results of experiments are given next.

6.1. Data sets

We generated synthetic data sets using the GSTD spatio-temporal data generator [25] following a scenario where cars move in a square region of 100 square kilometers. Each car is equipped with a GPS device, and transmits its position to the server at each timestamp using either radio communication links, or cellular phones. Cars are initially located around the left-bottom corner of the space, and are moving toward the center, and then gather around the right-upper corner. Such movement will change the distribution of cars from the initial skewed one to uniform and then skewed again. Cars are assigned a certain velocity with an equal probability, which is defined as the average distance to a specific direction that a car can move between two consecutive timestamps. We also assume that not all cars are active at each timestamp, that is, only a fraction (α percent) of cars change their positions between two consecutive timestamps. The α is called the *activity* of a set of cars or any moving objects.

For each data set, a set of data objects were created following the Zipf distribution within a unit square for the first timestamp. Then, at each of the subsequent timestamps, we randomly selected an α percent of the data objects and made them active with a certain velocity. The direction and velocity of an object can be controlled by properly adjusting the distributions of center coordinates.

There are three important parameters that affect the performance: (i) the distribution of data set, (ii) the activity of data set, and (iii) the average velocity of objects. We carried out experiments with a wide range of values for these parameters as follows. A Zipf parameter that controls the *skewedness* of data distribution was varied from 0 to 2. This parameter is the exponent z in a power-law function $P_i \sim \frac{1}{i^z}$. Note that the z value 0 means that the data set is uniformly distributed. The activity α was varied from 0 to 100. Notice that $\alpha = 0$ implies that no objects change its position between timestamps, whereas $\alpha = 100$ means that all objects change its position between timestamps. The average velocity of objects v was in ranges [0.0001, 0.32]. We selected the *adjustment* approach among three approaches provided by the GSTD to handle invalid objects fallen outside the work space. With these parameter settings, we generated various data sets with 10K objects each, evolving for 512 timestamps.

6.2. Settings for experiments

Experiments were performed on a Sun Ultra SPARC-IIi 333 MHz workstation running on Solaris 2.7. This workstation has 512MB of memory and 9.1GB of disk storage (IBM 18ES) with Ultra2 SCSI interface. We used the direct I/O feature of Solaris for all the experiments to avoid operating system's cache effects. Throughout the entire set of experiments, the same page size of 1KB was used for disk I/O. Using this page size, the fanouts of the HR-tree and the 3D R-tree were 42 and 36, respectively. For the MV3R-tree, the fanout was 36, $P_{version}$ and P_{svo} were set to 0.35 and 0.85, respectively. The fanout of the STR-tree was 28 and 36 for leaf and non-leaf nodes, and the fanout of the TB-tree was 31 and 36 for leaf and non-leaf nodes. The number of entries per page (E) of the $\mathcal{MO}$-**Cube** was 128. We divided the data space into 64 cells using Eq. (4) given in Section 4.1.1.

6.3. Time-slice/interval queries

In the first set of experiments, we compared the $\mathcal{AIM}$ with R-tree based methods for processing time-slice/interval queries. As already mentioned, we used data sets with cardinality 10K evolving for 512 timestamps. We generated square query windows whose side lengths are 10% or 20% of the unit [0,1). That is, the selectivity of each query was either one or four percent of the universe. The query windows were distributed uniformly in the universe $[0,1)^2$. In order to simulate real life situations, we executed workloads with 1000 queries for each sets of experiments. Cost was measured in terms of the average number of page accesses per query. In addition to that, actual query response times were measured.

Fig. 8(a) and (b) shows the average number of page accesses as a function of the interval when the spatial query selectivity was 1% and 4%, respectively. The length of temporal query intervals was varied between 0% and 20%. An interval of 0% corresponds to a single timestamp (i.e. a time-slice query), and an interval of 20% corresponds to approximately 100 timestamps. The notation used in this section is summarized in Table 2.

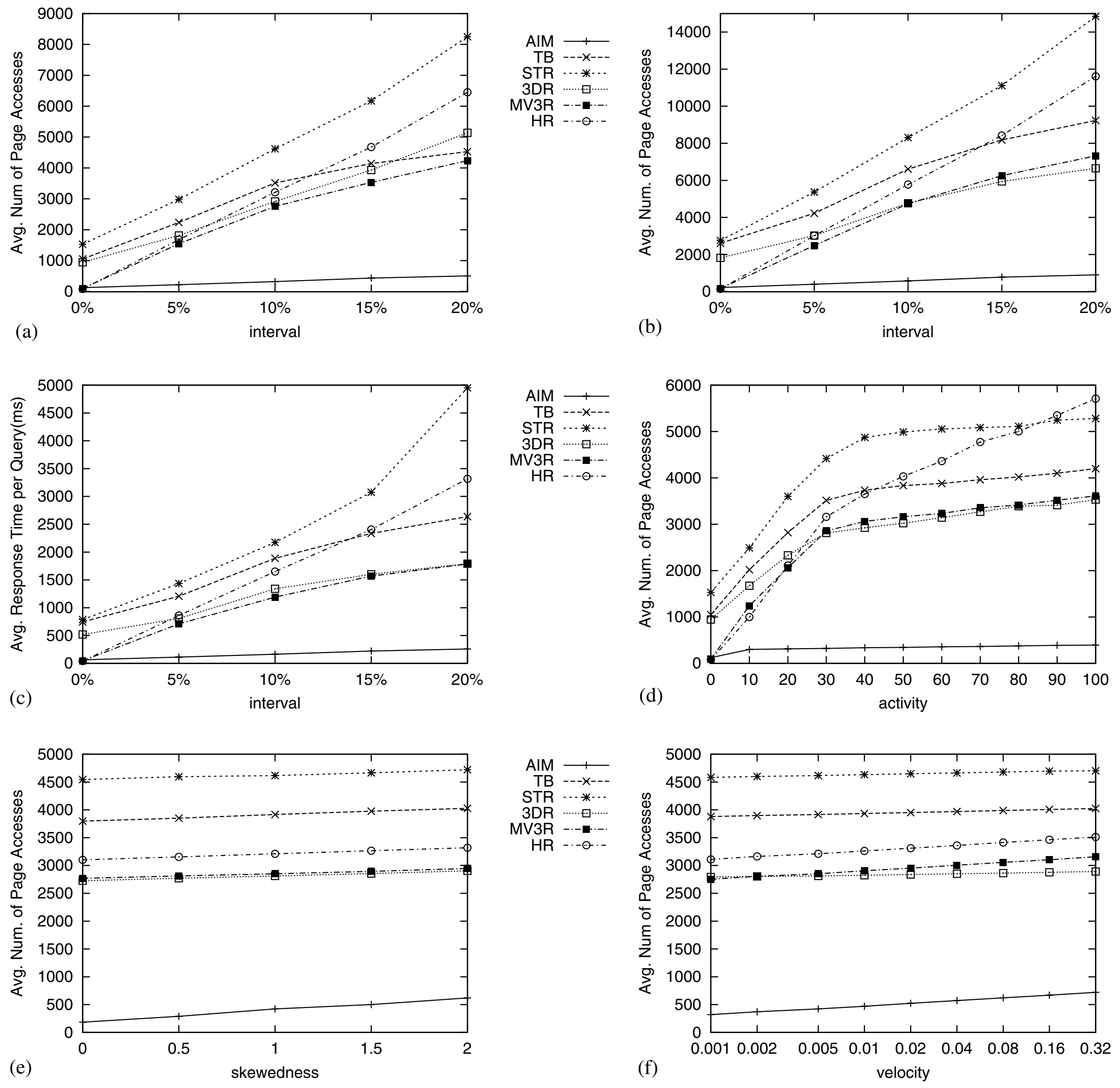
The $\mathcal{AIM}$ outperformed R-tree based methods except the extreme case of an interval of 0%. Regarding time-slice queries, the performance of the $\mathcal{AIM}$ was slightly worse than that of the HR-tree and the MV3R-tree, since the query process of the $\mathcal{AIM}$ included the *intersection test* (testing whether a line segment satisfies the query range), which needs to access the $\mathcal{MO}$-**Trace** per candidate obtained after range searches. However, for time-interval query (i.e., longer than 0%), The $\mathcal{AIM}$ consistently outperformed R-tree based structures. Fig. 8(c) shows the average elapsed time spent processing a time-slice/interval query. As Fig. 8(c) shows, the $\mathcal{AIM}$ outperformed R-tree based methods by a factor of two to three in average response time.

It is noteworthy that the six index structures were divided into two groups at an interval of 0% in Fig. 8(a)–(d) according to their performance behaviors. The 3D R-tree, the TB-tree and the STR-tree in the first group showed much higher processing cost for time-slice queries than the other group of index structures. This is due mainly to the fact that those in the first group are ephemeral index structures, whereas those in the second group (the $\mathcal{AIM}$, the HR-tree and the MV3R-tree) are multiple-index structures.

For an interval value below 10%, the MV3R-tree showed superior time-interval query performance over the 3D R-tree as shown in Fig. 8(b). For a longer, above 10%, this trend was reversed. However, as Fig. 8(c) shows, the MV3R-tree was slightly better than the 3D R-tree in average response time.

It is also important to notice that both the STR-tree and the TB-tree were worse than the 3D R-tree for interval values below 15% as shown in Fig. 8(a), since the TB-tree (The STR-tree) does not consider the spatial discrimination of the data sets at all (partially). However, for time-interval queries (longer than 15%), the TB-tree supports time-interval queries much more efficiently than the 3D R-tree does, since the TB-tree does "know" about the temporal discrimination of the data. On the other hand, in the case of the spatial query selectivity of 4%, the 3D R-tree show superior time-slice/interval query performance consistently over both the STR-tree and the TB-tree as shown in Fig. 8(b), since the spatial discrimination capabilities of the index became more important as the size of a query window increased.

In addition, we also executed extensive experiments to study how the behavior of the $\mathcal{AIM}$ and R-tree based methods vary according to activity, skewedness and velocity as shown in



a: activity, v: velocity, z: skewedness, q: spatial selectivity, I: temporal interval

Fig. 8. Time-slice/interval queries (a) $\alpha = 30$, $v = 0.005$, $z = 1$, $q = 1\%$ (b) $\alpha = 30$, $v = 0.005$, $z = 1$, $q = 4\%$ (c) $\alpha = 30$, $v = 0.005$, $z = 1$, $q = 1\%$ (d) $z = 1$, $v = 0.005$, $q = 1\%$, $I = 10\%$ (e) $\alpha = 30$, $v = 0.005$, $q = 1\%$, $I = 10\%$ (f) $\alpha = 30$, $z = 1$, $q = 1\%$, $I = 10\%$.

Fig. 8(d)–(f). First, Fig. 8(d) shows the effect of activity. The HR-tree was the most sensitive to the activity as shown in Fig. 8(d). The reason is that the reuse rate decreased rapidly as the activity increased. Meanwhile, the $\mathcal{AIM}$ was nearly unaffected by activity values, thus showed the best performance without regard to activity values.

Table 2
Notation for data sets and queries

Notation	Description	Values
a	Activity	[0, 100]
v	Velocity	[0.001, 0.32]
z	Skewedness	[0, 2]
q	Spatial query selectivity	[1%, 4%]
I	Temporal query interval	[0%, 20%]

All the index structures were slightly affected by the skewedness as shown in Fig. 8(e). Whereas the velocity showed non-trivial impact on query processing performance as shown in Fig. 8(f). In particular, it is important to notice that a multiversion and overlapping structure, i.e., the MV3R-tree, the HR-tree and the $\mathcal{AIM}$, were more sensitive to the velocity than the others.

6.4. Trajectory queries

For trajectory queries, we compared the $\mathcal{AIM}$ with the STR-tree and the TB-tree which are trajectory oriented originally. The 3D R-tree was also included in order to compare with the STR-tree and the TB-tree. Fig. 9 illustrates the performance of trajectory queries as a function of trajectory length. We used data sets with $\alpha = 30$, $v = 0.005$ and $z = 1$. As for the queries, we use three sets of the side length of cubic query window with 1%, 10% and 20% of the total range with respect to each dimension (i.e., x, y and t) and the length of trajectory needed to be retrieved after range search is in a range from 1% to 50%. Note that the trajectory length of 50% corresponds to 256 timestamps.

For trajectory queries, the $\mathcal{AIM}$ also consistently outperformed R-tree based methods. The performance measurements of the $\mathcal{AIM}$ were almost clung to the x axis. Although the TB-tree showed better performance than the 3D R-tree as shown in Fig. 9(a), its performance was worse than that of 3D-tree for trajectories of relatively short length as shown in Fig. 9(b) and (c). This is because the spatial discrimination becomes important in the case of short length trajectories. Thus, the 3D R-tree, which has much more space discrimination capabilities than the TB-tree, showed better performance than the TB-tree.

The performance of both the STR-tree and the 3D R-tree deteriorated rapidly as the length of trajectories increased. This fact shows that these methods have weak temporal discrimination capabilities. On the other hand, the curves of the $\mathcal{AIM}$ and the TB-tree were nearly parallel to x axis. This is because these methods strictly preserve trajectories by keeping line segments belonging to the same trajectory together.

Although the TB-tree processed trajectory queries much more efficiently than the 3D R-tree and the STR-tree in all but few exceptional cases, it was still consistently outperformed by the $\mathcal{AIM}$ as shown in Fig. 9(a)–(d).

6.5. Storage savings by reuse

In this subsection, we analyzed the reuse rates of the $\mathcal{AIM}$, the MV3R-tree and the HR-tree which adopt an *overlapping* technique or a *multiversion* structure. Fig. 10(a) shows the reuse rates of the $\mathcal{AIM}$, the MV3R-tree and the HR-tree. As expected, the reuse rates of the MV3R-tree and the

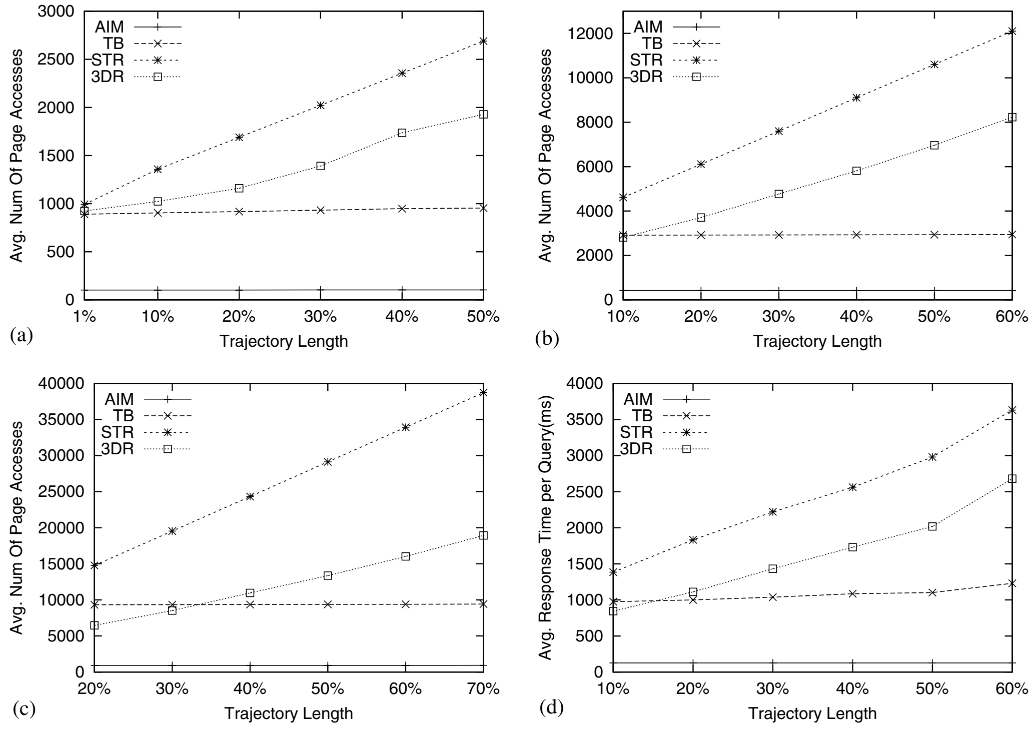


Fig. 9. Trajectory queries: (a) 1% in each dimension (x,y,t), (b) 10% in each dimension (x,y,t), (c) 20% in each dimension (x,y,t), (d) response time, 10% in each dimension.

HR-tree were much lower than that of the $\mathcal{AIM}$. Fig. 10(b) shows the reuse rates as a function of the activity. The reuse rates of the MV3R and the HR-tree decreased rapidly as the activity increased. This shows that the benefit from the reuse technique may not come up to our expectations, when more than 10% of the objects change their position at each timestamp. Fig. 10(c) illustrates the reuse rates as a function of the skewedness. Although the reuse rates slightly decreased as the skewedness increased, the reuse rate was less sensitive to the skewedness than the other factors.

To further analyze the reuse rate of the $\mathcal{AIM}$, we measured the reuse rate with varying velocities under different data distributions. Fig. 10(d) shows the reuse rates of the $\mathcal{AIM}$. In this experiment, the dominant factor for the reuse rate was the size of a cell. Specifically, if the width of a cell is smaller than the maximum distance an object can travel between two consecutive timestamps, it is highly probable that a new bucket is created. In case of a skewed data distribution, the average size of cells where more objects can be located tends to be smaller, due to more frequent refinements. This explains the fact that the reuse rate for the uniform distribution was higher than that for the skewed distribution, when the velocity was high.

6.6. Cost to build an index

Fig. 11(a) shows the size of various methods. As expected, the $\mathcal{AIM}$ had the smallest size. The size of the $\mathcal{AIM}$ was significantly smaller than that of R-tree based methods.

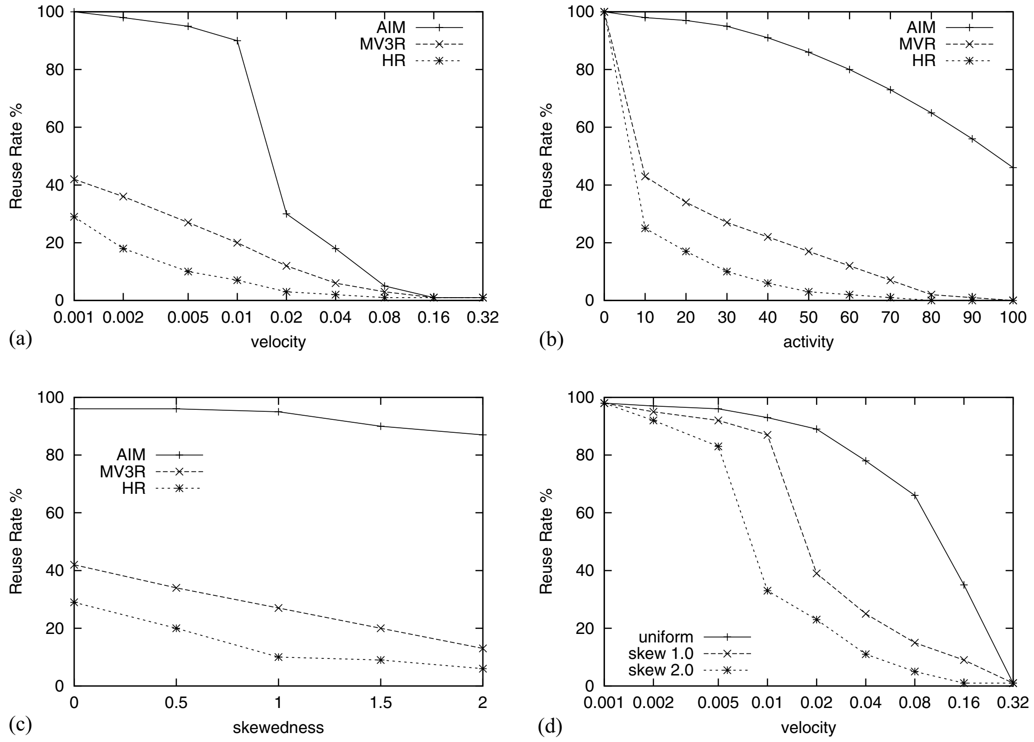


Fig. 10. Reuse rates: (a) $\alpha = 30$, $z = 1$; (b) $v = 0.005$, $z = 1$; (c) $\alpha = 30$, $v = 0.005$; (d) reuse rate of the $\mathcal{AIM}$.

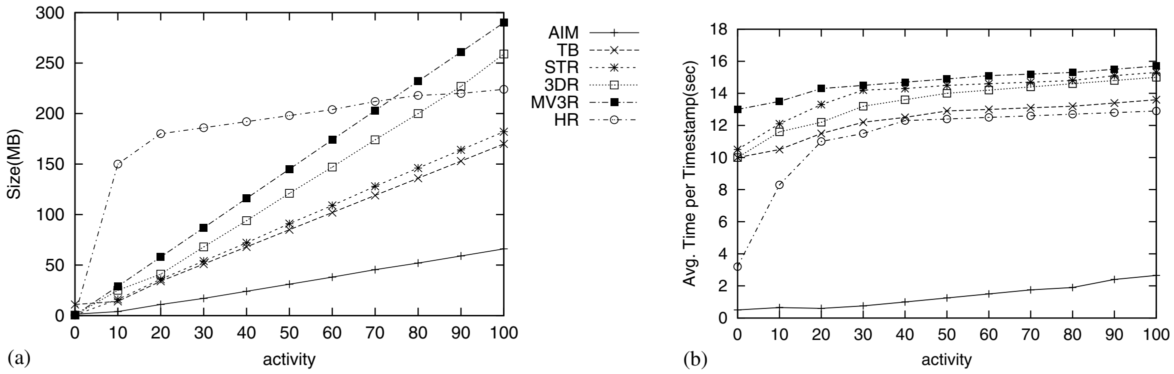


Fig. 11. Size and building time comparisons: (a) index size, (b) building time.

The size of the HR-tree increased rapidly when the activity was in range 0% from to 10%, which explains that the reuse rate was reduced rapidly. Although the size of the HR-tree was significantly larger than that of the other methods for less active data sets, it was smaller than those of the MV3R-tree and the 3D R-tree for highly active data sets. The sizes of the STR-tree and the TB-tree were smaller than that of the 3D R-tree since the average space utilizations of the SR-tree and the TB-tree is almost 100%. Notice that the size of the TB-tree at $\alpha = 0$ was about 11MB, because a leaf node only contained segments belonging to the same trajectory, resulting in 10K leaf nodes.

Next, we compared the $\mathcal{AIM}$ and R-tree based methods in terms of the wall clock time spent on building the indexes. Fig. 11(b) shows the average time spent to build an index per timestamp measured with varying activities. The building time of the $\mathcal{AIM}$ was significantly lower than those of R-tree based methods by about an order of magnitude. The HR-tree was the second best, despite the size was larger than that of the other methods. This is because that it is faster to build a new index for 10K objects than to insert them into an existing index with many objects. The building time of the MV3R-tree was slightly larger than that of the other methods. This is due to the time needed to manage the combination of the MVR-tree and the 3D R-tree and the cost of the re-insertion function.

In summary, the $\mathcal{AIM}$ not only has the smallest index size, but also requires the least time to build. This results demonstrate that the $\mathcal{AIM}$ is much better suited for mission-critical applications with high time and space complexities.

7. Conclusions and future work

This is the first study that adopts a cell-based index structure for the current and historical information of moving objects, instead of hierarchical access methods such as R-trees. We have proposed an adaptive cell-based approach, called $\mathcal{AIM}$, which consists of two main structures: $\mathcal{MO}$-**Cube** and $\mathcal{MO}$-**Trace**. The $\mathcal{MO}$-**Cube** utilizes an *overlapping* technique to reduce the storage requirements, and refines cells adaptively to handle data skew problems with only very small space overhead. The $\mathcal{MO}$-**Trace** strictly preserves the entire trajectories of moving objects in a space-efficient way.

Through the extensive experiments, the $\mathcal{AIM}$ has proven to be an effective access method well suited for moving objects. The $\mathcal{AIM}$ achieved reductions in storage requirement up to an order of magnitude more than those by R-tree based approaches. In addition, the $\mathcal{AIM}$ outperformed R-tree based approaches consistently and considerably for both time-slice/interval queries and trajectory queries.

The $\mathcal{AIM}$ can be augmented with the capability of predicting the future locations of moving objects. We plan to work toward the direction and also plan to develop algorithms for k-nearest neighbor searches with the $\mathcal{MO}$-**Cube**. Finally, we also expect the $\mathcal{AIM}$ to be applied to spatio-temporal data warehouses for moving objects.

Acknowledgements

This work was supported in part by the Brain Korea 21 Project and the Information Technology Research Center (ITRC) Support Program from the Government of Korea. It was also sponsored in part by National Science Foundation CAREER Award (IIS-9876037), Grant No. IIS-0100436, and Research Infrastructure program EIA-0080123. The authors assume all responsibility for the contents of the paper.

References

[1] Tamas Abraham, John F. Roddick, Survey of spatio-temporal databases, GeoInformatica 3 (1) (1999) 61–99.

[2] Pankaj K. Agarwal, Lars Arge, Jeff Erickson, Indexing moving points, in: Proceedings of the 19th ACM Symposium on Principles of Database Systems, Dallas, TX, May 2000, pp. 175–186.

[3] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, Peter Widmayer, An asymptotically optimal multiversion B-tree, VLDB Journal 5 (4) (1996) 264–275.

[4] Jon Louis Bentley, Donald F. Stanat, E. Hollings Williams Jr., The complexity of finding fixed-radius near neighbors, Information Processing Letters 6 (6) (1977) 209–212.

[5] F. Warren Burton, Matthew M. Huntbach, John G. Kollias, Multiple generation text files using overlapping tree structures, The Computer Journal 28 (4) (1985) 414–416.

[6] F. Warren Burton, John G. Kollias, D.G. Matsakis, V.G. Kollias, Implementation of overlapping B-trees for time and space efficient representation of collection of similar files, The Computer Journal 33 (3) (1990) 279–280.

[7] Hae Don Chon, Divyakant Agrawal, Amr El Abbadi, Using space–time grid for efficient management of moving objects, in: ACM International Workshop on Data Engineering for Wireless and Mobile Access, 2001.

[8] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, Markus Schneider, A data model and data structures for moving objects databases, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000, pp. 319–330.

[9] Volker Gaede, Oliver Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.

[10] Antonin Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, June 1984, pp. 47–57.

[11] George Kollios, Dimitrios Gunopulos, Vassilis J. Tsotras, On indexing mobile objects, in: Proceedings of the 18th ACM Symposium on Principles of Database Systems, Philadelphia, PA, May 1999, pp. 261–272.

[12] Dongseop Kwon, Sangjun Lee, Sukho Lee, Indexing the current positions of moving objects using the Lazy update R-tree, in: Proceedings of the 3rd International Conference on Mobile Data Management, Singapore, January 2002, pp. 113–120.

[13] David B. Lomet, Betty Salzberg, Transaction-time databases, in: Temporal Databases, 1993, pp. 388–417.

[14] Yannis Manolopoulos, G. Kapetanakis, Overlapping B^+-tree for temporal data, in: Proceedings 5th Jerusalem Conference on Information Technology, Jerusalem, Israel, 1990, pp. 491–498.

[15] Mario A. Nascimento, Jefferson R.O. Silva, Towards historical R-trees, in: Proceedings of the 1998 ACM Symposium on Applied Computing, Atlanta, GA, February 1998, pp. 235–240.

[16] J. Nievergelt, H. Hinterberger, K.C. Sevcik, The grid file: an adaptable, symmetric multi-key file structure, ACM Transactions on Database Systems 9 (1) (1984) 38–71.

[17] Dieter Pfoser, Christian S. Jensen, Yannis Theodoridis, Novel approaches in query processing for moving object trajectories, in: Proceedings of 26th International Confernce on Very Large Data Bases, Cairo, Egypt, September 2000, pp. 395–406.

[18] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, Mario A. Lopez, Indexing the positions of continuously moving objects, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, May 2000, pp. 331–342.

[19] Hanan Samet, The quadtree and related hierarchical data structures, ACM Computing Surveys 16 (2) (1984) 187–260.

[20] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, Son Dao, Modeling and querying moving objects, in: Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham, UK, April 1997, pp. 422–432.

[21] Zhexuan Song, Nick Roussopoulos, Hashing Moving Objects, In Proceedings of the 2nd International Conference on Mobile Data Management, pages 161–172, Hong Kong, China, January 2001.

[22] Yufei Tao, Dimitris Papadias, MV3R-tree: a spatio-temporal access method for timestamp and interval queries, in: Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy, September 2001.

[23] Jamel Tayeb, Özgür Ulusoy, Ouri Wolfson, A quadtree-based dynamic attribute indexing method, The Computer Journal 41 (3) (1998) 185–200.

[24] Yannis Theodoridis, Timos K. Sellis, Apostolos Papadopoulos, Yannis Manolopoulos, Specifications for efficient indexing in spatiotemporal databases, in: 10th International Conference on Scientific and Statistical Database Management, Capri, Italy, July 1998, pp. 123–132.
[25] Yannis Theodoridis, Jefferson R.O. Silva, Mario A. Nascimento, On the generation of spatiotemporal datasets, in: Proceedings of the 6th International Symposium on Spatial Databases, 1999, pp. 147–164.
[26] Yannis Theodoridis, Michalis Vazirgiannis, Timos K. Sellis, Spatio-temporal indexing for large multimedia applications, in: Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems, Hiroshima, Japan, 1996, pp. 441–448.
[27] Theodoros Tzouramanis, Michael Vassilakopoulos, Yannis Manolopoulos, Overlapping linear quadtrees: a spatio-temporal access method, in: Proceedings of the 6th ACM International Workshop on Geographical Information-Systems, Washington, DC, November 1998, pp. 1–7.
[28] Ouri Wolfson, Bo Xu, Sam Chamberlaina, Liqin Jiang, Moving objects databases: issues and solutions, in: Proceedings of 10th International Conference on Scientific and Statistical Database Management, 1998, pp. 111–122.
[29] X. Xu, J. Han, W. Lu, RT-tree: an improved R-tree index structure for spatiotemporal databases, in: Proceedings of the 4th Intl. Symposium on Spatial Data Handling, SDH'90, Zurich, Switzerland, 1990, pp. 1040–1049.

Wonik Choi is a Ph.D. student in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea. His current research interests include spatio-temporal databases, mobile databases, geographic information systems, XML. He received his MS and BS degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1998 and 1996, respectively.

Bongki Moon is an Associate Professor of Computer Science at the University of Arizona. His current research interests include high performance spatial and temporal databases, scalable web servers, data mining and warehousing, and parallel and distributed processing. He received his Ph.D. degree in Computer Science from University of Maryland, College Park, in 1996, and his MS and BS degrees in Computer Engineering from Seoul National University, Korea, in 1985 and 1983, respectively. He was a member of the research staff at Communication Systems Division, Samsung Electronics Corp., Korea, from 1985 to 1990. He received the National Science Foundation CAREER Award in 1999.

Sukho Lee received his BA degree in Political Science and Diplomacy from Yonsei University, Seoul, Korea, in 1964 and his MS and Ph.D. in Computer Sciences from the University of Texas at Austin in 1975 and 1979, respectively. He is currently a professor of the School of Computer Science and Engineering, Seoul National University, Seoul, Korea, where he has been leading the Database Research Laboratory. He has served as the president of Korea Information Science Society. His current research interests include database management systems, spatial database systems, and multimedia database systems.